TAG: Advancements in AI-Driven Tabletop Games

Tutorial @ IEEE CoG'23

-

Running TAG (Tournaments and Advanced Agents)

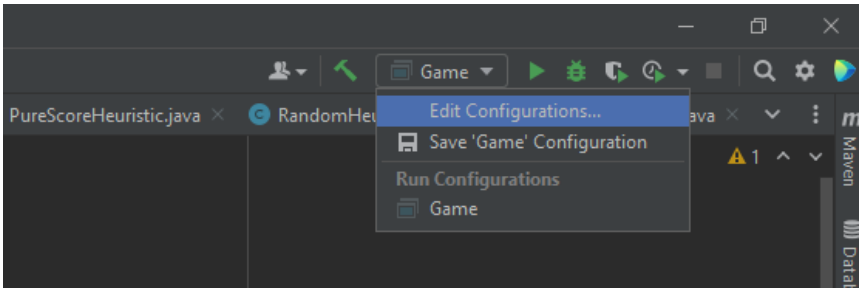
1 Running The Framework for AI Agents (Arguments and Round Robin)

We've seen so far one basic way of running the TAG framework, through the `Game` class (either running one game or several). There are different ways of running TAG that are more useful for an empirical and experimental approach. For Game AI agents, more detailed information from a run can be obtained by running the framework using the `evaluation.RunGames` class. This class facilitates running it via command line or with program arguments from an IDE, which we'll explore first.

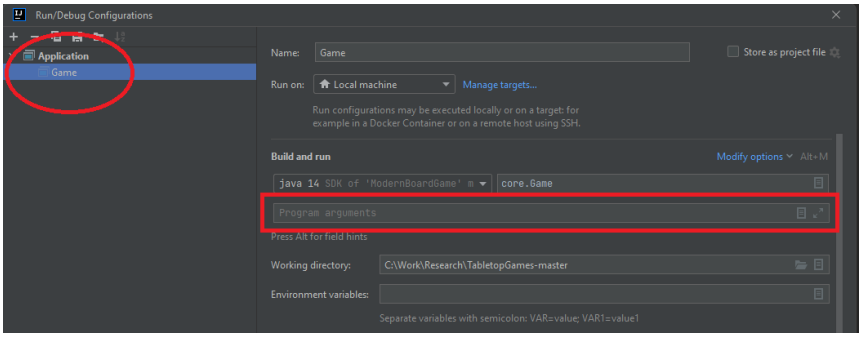
1.1 Program Arguments in Intellij IDEA

In order to run a Java application **with arguments** in Intellij IDEA, you need to follow the next steps[1]:

1. Open the Edit Configuration window, which you can find in the upper right side of the IDE:



2. This window allows you to set up what happens when you run the application. Note that you may have several configurations (see the left side) set up — if not, create a new configuration, of type "Application". The window shows the name of the configuration, the Java JDK that is used to build the code, the class that is executed and the program arguments (highlighted in the image below):



3. Arguments are added to the program in the "Program Arguments" field. TAG is implemented so you don't have to worry about reading this arguments, but you can write them here when you run the Round Robin Tournament functionality.

[1] You can also check the official documentation at `https://www.jetbrains.com/help/idea/running-applications.html#customizable-way`.

1.2 Round Robin Tournaments

While *runMany()* allows you to quickly run several players in several games, the `evaluation.RunGames` class provides a more convenient way to run a Round Robin Tournament between multiple AI players for a single game, for N repetitions. This is useful to run multiple games and extract statistics of performance. The RunGames class allows you to set up all agents to play against each other in one or more games. You can specify:

- The game
- The number of players
- The directory containing a set of JSON files, one for each of the agents to be included in the tournament. Details here: `http://www.tabletopgames.ai/wiki/agents/agent_JSON`.
- Whether to run in 'exhaustive' or 'random' mode. The former runs a number of games for every possible player combination across every possible position. The latter runs a total number of games, randomising the players (and their positions) in each - it is more tractable for larger numbers of players, and/or if each game takes a long time to run.
- A set of Metrics to report interesting data on the games run. Details here: `http://www.tabletopgames.ai/wiki/running/IGameListener`.

For full details on usage use the `--help` option on the command line (as described above, write "–help" in the program arguments field in IntellijIDEA). You can also check here `http://www.tabletopgames.ai/wiki/running/Tournament` for instructions on the different parameters of this class. However, the easiest way to configure a tournament is via a JSON configuration file. An example of the contents of such a file can be seen below:

```
{
    "game" : "Connect4",
    "nPlayers" : 2,
    "mode" : "exhaustive",
    "matchups" : 100,
    "verbose" : false,
    "seed" : 740234
}
```

Each one of the lines above indicates one parameter for the round robin tournament. This includes the game, the number of players in the game, the number of games to be played, etc. Note that these parameters can be set via command line, but you can also modify the default values in the JSON file. All these parameters take default values (again, check here: `http://www.tabletopgames.ai/wiki/running/Tournament`) if they're not specified.

In order to run a tournament with the above configuration, you need to indicate, via arguments, the location of the JSON file. For instance, if the above JSON is in the file *json/experiments/rungames0.json*, you must an argument to the IDEA configuration (as seen in the previous section) like this:

```
config=json/experiments/rungames0.json
```

Exercise 1 Try to do this now: add the above line as an argument to the execution to the RunGames class and run the application. Observe the output that is generated when the tournament finishes (it may take 1 or 2 minutes; if your machine is slow, simply reduce the value of "matchups" in the JSON file).

As you can see, a lot of output is generated. This is providing information about how did the games go during the tournament. To facilitate analysing the output, we can add more parameters to the JSON file to configure how results are presented. The next JSON adds 3 parameters that define a Listener and the directory for the tournament output:

```
{
    "game" : "Connect4",
    "nPlayers" : 2,
    "mode" : "exhaustive",
    "matchups" : 100,
    "verbose" : false,
    "seed" : 740234
```

```
    "listener" : "json/listeners/basiclistener.json",
    "destDir" : "outputdir",
    "output" : "outputdir/Tournament.log",
}
```

Exercise 2 Try running this now. Create a new JSON file with these new arguments and modify the execution argument so the new configuration file is picked up. Note that you will also have to create the directory specified in the "destDir" parameter. Then run the application and observe that no output is generated to the console, but files with results have been created in the output directory specified. Take a look at those!

One thing that may have caught your attention is the sub-directories generated in the output directory: this is for each pair of agents that took part in the tournament. At this point, you may be wondering how can you indicate which AI agents are included in the run.

First of all, as we did not indicate any agents in our JSON file, TAG ran the tournament with the "default" set of agents, which is basically whatever is written in the code. If you look at the code in RunGames, main() function, you'll find the following set of lines:

```
LinkedList<AbstractPlayer> agents = new LinkedList<>();
if (!runGames.config.get(playerDirectory).equals("")) {
    agents.addAll(PlayerFactory.createPlayers((String) runGames.config.get(
        playerDirectory)));
} else {
  /* 2. Dafault players HERE */
  agents.add(new MCTSPlayer());
  agents.add(new BasicMCTSPlayer());
  agents.add(new RandomPlayer());
  agents.add(new RMHCPlayer());
  agents.add(new OSLAPlayer());
}
```

These above are the agents that have been run, because the previous JSON file did not specify any. However, specifying agents by code is more verbose, prone to error and slower than using JSON. In order to specify the agents via JSON, we only have to add another parameter to the file (*playerDirectory*):

```
{
    "game" : "Connect4",
    "nPlayers" : 2,
    "mode" : "exhaustive",
    "matchups" : 100,
    "verbose" : false,
    "seed" : 740234,
    "listener" : "json/listeners/basiclistener.json",
    "destDir" : "outputdir",
    "output" : "outputdir/Tournament.log",
    "playerDirectory" : "tournamentPlayers"
}
```

Note that this is a player directory, not a JSON file. This is because we indicate each agent in a different JSON file, and all are stored in the same directory. A full description of what to include in agent JSON configuration files can be found here: `http://www.tabletopgames.ai/wiki/agents/agent_JSON`. Here are three examples of agents that can be run in TAG (Random, First Action Player and One-Step Look Ahead). You can also find these json files in the *json/players* directory.:

```
{
  "class" : "players.simple.RandomPlayer"
}
```

```
{
  "class" : "players.simple.FirstActionPlayer"
}
```

```
{
  "class":"players.simple.OSLAHeuristic",
  "heuristic" : {
    "class" : "players.heuristics.ScoreHeuristic",
    "maxLoot" : 500,
    "BULLET_CARDS" : -1.0,
        "BULLETS_PLAYER" : 0.5,
    "BULLETS_ENEMY" : 0.0,
    "LOOT" : 0.5
  }
}
```

Exercise 3 Let's run a round robin tournament where 3 agents (Random, OSLA and FirstActionPlayer) play the game *Connect 4* for 100 repetitions. Do the following:

- Create a new JSON configuration file for the tournament that indicates the game, number of players (2), number of games played (100), the tournament mode (exhaustive), the listener file (same as in the examples above), output and player directories, and a name for the output log file.
- Make sure the output directories does not exist or it's empty (otherwise the previous results will be overwritten)
- Create a new directory with the name specified in your JSON configuration file and add one JSON file for each one of the agents, with the text specified above (you can also copy the files from *json/players*).

Run the RunGames class once everything is set up. It'll take a bit to complete the execution but the output will show the aggregated results of all games played. Observe the results in the output directory:

- The output file indicates, in sequence, the results of the games played by each pair of agents. Note that each result is an aggregate of 100 games, as indicated in the configuration. Note the following
  - The "points" account for the number of games won during the pairing
  - Each player has a win rate for each pairing.
  - Each pair of players has its own ranking, which specifies the win rate of each one of them.
- The output directory also contains several sub-directories, which show all the individual games played during the tournament (per pair of players).
- A global ranking is included at the end of the file, with win rates obtained by each player.

2 Game States and Forward Models

The core classes implemented by all games in the framework are the **game state** and the **forward model**. We will explore implementations of these for some of the games next.

2.1 Game State

All game states for all games extend from `core.AbstractGameState`, which provides a lot of basic functionality and templates for further functionality (i.e. methods which can be implemented by subclasses). A game state contains all information needed to describe the current state of the game at a moment in time: components, other variables or groupings. It does not implement any game functionality, it is simply a container class. Any functions implemented are simple accessor functions, to give wider access to the information contained.

The state is going to be modified by the forward model (game rules) and player actions, which is where the main functionality for a game is actually implemented.

The most important method on a game state is the `copy` method, which has two purposes:

1. It creates *deep* copies of the game state: AI players can then make simulations within copies without affecting game objects in the original state.

2. If the game is partially observable (some information is hidden from the players, e.g. cards in hand), this method should hide components not visible by the player passed as an argument to this function. Note that this argument could be -1, in which case nothing in the state will be hidden.

2.2 Forward Model

The forward model encompasses the rules of a game. A copy of this model (with a different random seed) is given to all AI players, allowing them to make simulations of games in parallel to the real game being played, following the same rules. All forward models extend from the `core.AbstractForwardModel` class (similar to the game state, this provides functionality and templates for implementations of new games).

The functionality is split across several important functions:

- **Setup:** Sets up the initial state of the game.
- **Next:** applies the main game rules when player decisions are taken. It takes care of the following:
  1. Apply the given action to the game state.
  2. Execute any other required game rules (e.g. change the phase of the game).
  3. Check for game end.
  4. Move to the next player (if required, and if the game has not ended).
- **Compute Available Actions:** returns a list of all actions available in the game state passed as an argument, for the next player to pick from.

Game Example: Tic Tac Toe In Tic Tac Toe[2], 2 players alternate placing their symbol in a N $\times$ N grid until one player completes a line, column or diagonal and wins the game; if all cells in the grid get filled up without a winner, the game is a draw. This is the simplest game included in the framework, meant to be used as a quick reference for the minimum requirements to get a game up and running. The game Tic Tac Toe is implemented in the package *games.tictactoe*.

The **Forward Model** implements the required functions as follows, in the class *games.tictactoe.TicTacToeForwardModel.java*:

- **Setup:** sets up the grid board, according to the size defined in the games parameters (more on this next week). See the method `_setup(AbstractGameState)`.
- **Next:** See the method `_next(AbstractGameState, AbstractAction)`.
  1. Applies the given action to the game state, filling in one of the grid cells with the current player's associated symbol.
  2. Checks for game end (a line filled with the same player's symbol, or the grid filled with no winner), and assigns game status and player result accordingly.
  3. If the game has not ended, moves to the next player.
- **Compute Available Actions:** actions in this game are to place the player's symbol in an empty cell on the grid, so a list of all such actions is returned for the next player to choose from. See the method `_computeAvailableActions(AbstractGameState)`.

The **Game State** is implemented in *games.tictactoe.TicTacToeGameState*, and the copy method is named `_copy()`. This function creates a new object of type GameState identical to the object `this`.

Exercise 4 Inspect and try to understand the contents of the Game State and Forward Model functions indicated in the previous section (copy, setup, next, compute actions), for two games:

1. The game Tic Tac Toe, in the files indicated above.
2. The game Connect 4[a]. Connect 4 is implemented in the package `games.connect4`, and the game state and forward model classes are, respectively, `Connect4GameState` and `Connect4ForwardModel`.

[2]More information about this game here: `https://en.wikipedia.org/wiki/Tic-tac-toe`

Ask in class if any of the code in these function is not clear.

[a] More information about this game here: `https://en.wikipedia.org/wiki/Connect_Four`

3 Simple Agents

3.1 Simple agent: First Action Player

In TAG, every agent in the framework has to extend the `core.AbstractPlayer` class and implement the *_getAction()* and *copy()* methods. For instance, possibly the simplest agent in the framework is the `players.simple.FirstActionPlayer` player which, as its name indicates, simply executes the first available action as given by the game. Its complete code is shown here:

```
1  package players.simple;
2
3  import core.AbstractGameState;
4  import core.actions.AbstractAction;
5  import core.AbstractPlayer;
6  import java.util.List;
7
8  public class FirstActionPlayer extends AbstractPlayer{
9
10     @Override
11     public AbstractAction _getAction(AbstractGameState observation, List<
           AbstractAction> possibleActions) {
12         return possibleActions.get(0);
13     }
14
15     @Override
16     public FirstActionPlayer copy() {
17         return this;
18     }
19
20     @Override
21     public String toString() {
22         return "FirstAction";
23     }
24 }
```

The _getAction() method is called by the game every time this player needs to play an action in the game. It receives two parameters:

- ***observation***: An object of type `core.AbstractGameState`. This object encapsulates what the player can see in the current game. This observation includes, for example, common elements like the board, and also private elements like its own hand of cards.
- A list of available ***actions***: when calling the _getAction() method, the game provides the agent with a list of available actions. The order of this actions in the array is arbitrary, and it's determined by the concrete implementation of the forward model for the game being played.

The function copy() receives no parameters and must return a copy of this agent[3]. As can be seen in the code snippet above, another function is implemented: toString(). This method overrides a function from the parent class, AbstractPlayer, but it is no required for the agent to function. The possible functions that can also be overriden from the parent class are (see `core.AbstractPlayer` for their signatures):

- *initializePlayer*: called at the start of the game with a copy of the initial game state. Override to initialize the agent before the game starts[4].
- *finalizePlayer*: called once the game is over, with a copy of the final game state. Override to take care of different things (i.e. logging, or updating weights for an AI learning algorithm) once the game has concluded.

[3] FirstActionPlayer is so simple that simply returning "this" works, as nothing can change in the object that requires the creation of a new object. For an actual copy example, see the Random agent snippet below.

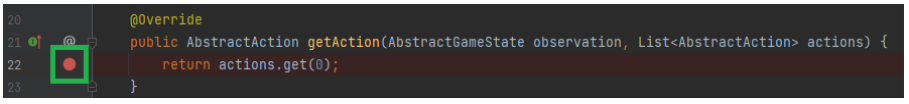
[4] For runs were multiple games are played, it's important to do this properly to reset any data you don't want to carry from one game run to the next (e.g. a record of which cards have been played already).

- *toString*: provides a custom name for the player (by default, if method not overriden, it'll be the name of the class).
- *registerUpdatedObservation*: in some cases the player only has a single or no actions to choose from (think of a game where you have no more actions to play but others are still taking turns). In these cases, instead of calling _getAction(), the game calls registerUpdatedObservation(). This function lets the player know of the current game state, so it can update its beliefs and internal members accordingly.
- *setForwardModel*: sets the forward model the agent can use to roll states forward. Only override if you need to perform an operation at this time.
- *getDecisionStats*: override this to provide information on the last decision taken (this is normally used by the GUI, so you'll likley be able to ignore this).

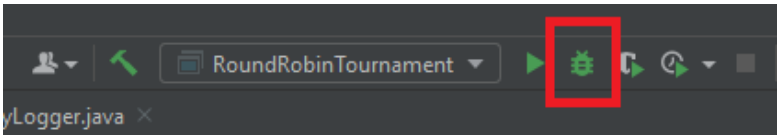
3.2 Debugging in IntellijIDEA

One of the main strengths of using an IDE for programming is the capactity to debug your code. In essence, Intellij IDEA (as well as other IDEs), permits you to pause the execution of your application at any instruction and inspect the values of the different variables at that time. Besides, you can also advance the exectution step by step from that point, and even *go in* functions when these are called.

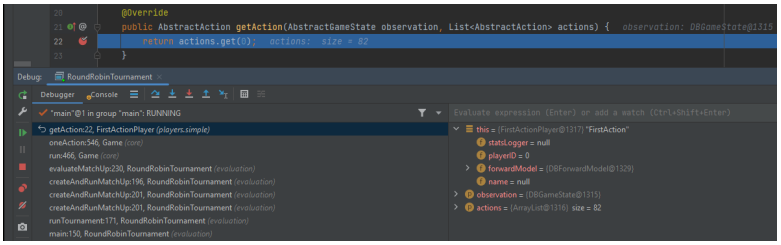
You can set breakpoints to indicate where you want your execution to stop, by clicking on the right of the line number:



The run will stop here (assuming the execution flow will actually traverse that point) if you run your code in debug mode, which you can do by clicking on the green bug icon on the top right:



When this happens, you can see how the execution halts and two panels are display at the bottom. There is a call stack (on the left in the image below) and a variable inspector (on the right), where you can see the values of the accessible variables at this point and also evaluate new expressions by writing them in this panel:



Finally, there are several commands that allow you to continue the execution:

- F7: Step Into (executes one instruction going into the next function in the current line).
- F8: Step Over (executes the following instruction in sequence, without entering functions called from the current line of code).
- F9: Resumes the execution.

Exercise 5 Set a breakpoint in the (only) line of code inside getAction() function of the FirstActionPlayer class. Then, modify the round robin tournament set of agents so FirstActionPlayer is included in the list, set up the game to be *TicTacToe*, and run (in Debug mode) a tournament. By inspecting the variables in the debugger, see if you can find the following:

- The size of the action space for the current state. These actions are stored in the list `actions` received

by parameter.

- The value of the property playerID of this player. This is stored in the reference `this` (i.e. it's the object whose function is being executed).
- Take a look at the observation variable, which is also received by parameter. This is the observation of the current game state. Here you can find information about the game type, the grid size (in game parameters) and - perhaps most importantly - the current state of the board (`observation.gridBoard` variable).
- Write 'actions.get(0)' (notice that this is what the function getAction() returns) in the "Evaluate expression ..." field in the variable inspector panel, and press Enter. A new item should appear in the list of variables with the result of your query. If you inspect this object (of type `SetGridValueAction`) you'll be able to see the coordinates x and y of the grid where this player will make a move.

3.3 Simple agent: Random Player

Another simple agent is the Random player, which returns an action uniformly at random from the list of available actions. It looks like this:

```
1  public class RandomPlayer extends AbstractPlayer {
2
3      //Random generator for this agent.
4      private final Random rnd;
5
6      //Constructor with a random generator for this agent.
7      public RandomPlayer(Random rnd) {
8          this.rnd = rnd;
9      }
10
11     // Default constructor for this agent.
12     public RandomPlayer() {
13         this(new Random());
14     }
15
16     // Get Action: this is called every action from the agent is needed.
17     // Returns an action to be executed in the game.
18     @Override
19     public AbstractAction _getAction(AbstractGameState observation, List<
           AbstractAction> actions) {
20         int randomAction = rnd.nextInt(actions.size());
21         return actions.get(randomAction);
22     }
23
24     // Just provides a "printable" name for this agent.
25     @Override
26     public String toString() {
27         return "Random";
28     }
29
30     // Creates a copy of this agent.
31     @Override
32     public RandomPlayer copy() {
33         return new RandomPlayer(new Random(rnd.nextInt()));
34     }
35 }
```

Exercise 6 Take a look at the _getAction() and copy() methods of the random agent and see if you can understand what they're doing (ask if you don't!).

3.4 Simple agent: One-Step Look Ahead

One-Step Look Ahead (or OSLA) is another simple agent that makes decisions based only on the quality of the state reached *after* applying each one of the possible actions. In order to achieve this, the OSLA agent (`players.simple.OSLAPlayer`) requires two elements:

- A Forward Model to roll states forward supplying an action.
- A Heuristic to evaluate a game state.

The players owns a reference to the **forward model** of the game, which can be accessed with the function *getForwardModel()*. Therefore, to advance the game state, the agent can do:

```
1 getForwardModel().next(gameState, act);
```

where 'gameState' is the game state where we want to play an action, and 'act'. is the actual action we want to play. Note this call **modifies** the object 'gameState', by *rolling* it forward after 'act' has been applied. Therefore, you could do something like:

```
1 getForwardModel().next(gameState, act);
2 float val = evaluate(gameState);
```

to evaluate how good or bad playing action 'act' in the game state 'gameState' resulted. Here, evaluate() would be a function implemented in the agent, which would be closely related to the **heuristic** the agent uses to evaluate states[5]. The following snippet shows code that iterates through all available actions, uses the forward model to roll states with each one of them and obtains a value for future states using a heuristic:

```
1 public AbstractAction _getAction(AbstractGameState gs, List<AbstractAction> actions)
    {
2   int playerID = gs.getCurrentPlayer();
3   for (int actionIndex = 0; actionIndex < actions.size(); actionIndex++) {
4     AbstractAction action = actions.get(actionIndex);
5     AbstractGameState gsCopy = gs.copy();
6     getForwardModel().next(gsCopy, action);
7     double val = heuristic.evaluateState(gsCopy, playerID);
8   }
9 }
```

There are some interesting things about this code:

- **Player ID**: each player in TAG has an ID, which can be retrieved with the AbstractPlayer function getPlayerID(). At the same time, a game state has a current player ID: this is, the ID of the player whose turn it is. Note that these may or may not be the same. If 'gs' is the game state received in the _getAction() function, player.getPlayerID() will be the same as gs.getCurrentPlayer(): it's this player's turn to play. However, as game states are rolled forward using the forward model, turns will progress to other players, and gameState.getCurrentPlayer() will return IDs different to the one of our agent. In OSLA, we don't care about this, because we only roll the state forward once (*rollout length* is 1). But other agents may want to deal with this factor.
- Line 6 **copies** the game state. This is done to make sure that each time the forward model is used, we are advancing the *copy* of the game state received by parameter in the function. We do not want to keep applying actions *consecutively* (rollout length > 1) over the same state, but applying each action to the *original* game state, one by one.
- **heuristic.evaluateState** takes care of determining the value of the action taken. We'll talk about heuristics later, but for know just think of this as a function that indicates how good a state is for a given player (given the ID of that player) with a numeric value. Normally, higher numbers indicate a better playing position for the player whose ID is indicated.

The only aspect missing is to identify the action to execute. The key is to determine for which one the actions tried, the heuristic evaluation provides the higher value. The following code incorporates that aspect:

```
1 public AbstractAction _getAction(AbstractGameState gs, List<AbstractAction> actions)
    {
2   double maxQ = Double.NEGATIVE_INFINITY;
3   AbstractAction bestAction = null;
4   double[] valState = new double[actions.size()];
5   int playerID = gs.getCurrentPlayer();
6
7   for (int actionIndex = 0; actionIndex < actions.size(); actionIndex++) {
```

[5]There are many considerations on this when working with stochastic and/or partially observable games: for instance, advancing a state with an action that has a random element (i.e. rolling a die) may lead to different outcomes every time is played. Or, if the game state contains incorrect assumptions you've made of your opponent cards, the evaluation may be completely off. That is the hard (and fun!) part of writing good AI agents for this type of games.

```
8       AbstractAction action = actions.get(actionIndex);
9       AbstractGameState gsCopy = gs.copy();
10      getForwardModel().next(gsCopy, action);
11      valState[actionIndex] = heuristic.evaluateState(gsCopy, playerID);
12
13      double Q = noise(valState[actionIndex], this.epsilon, this.random.nextDouble());
14
15      if (Q > maxQ) {
16        maxQ = Q;
17        bestAction = action;
18      }
19    }
20    return bestAction;
21  }
```

What's new in this code?

- All state values are stored in an array (`valState`).
- Action values are "filtered" through the function `noise()`. This function introduces a small **random** perturbation (of the order of 10^{-6}) to the value returned by the heuristic evaluation. This is done so two evaluations that return the same value can be differentiated by a small amount (in other words, it essentially allows to establish a preference, uniformly at random, between two actions that seem equally good/bad).
- `maxQ` and `bestAction` store the best value and action found so far. The latter is returned at the end of the function to be played in the game.

Exercise 7 Set up a breakpoint on the first line of the _getAction() function of the One Step Look Ahead. Make sure the RandomPlayer is still in the list of agents that would play a tournament and that you're still running Tic Tac Toe as the game.

Run in Debug mode and, using the variable inspector and the commands to advance and enter the different instructions of the method (F7 and F8), see if you can understand what the code is doing. If you don't, ask!

3.4.1 Heuristics

As you may have deduced, the heuristic evaluation of a game state is very important to determine the value of the actions that can be played and, consequently, the strength of the agent. TAG provides an interface for defining state heuristics: core.interfaces.IStateHeuristic, which looks like this:

```
1  package core.interfaces;
2  import core.AbstractGameState;
3
4  public interface IStateHeuristic {
5
6      /**
7       * Returns a score for the state that should be maximised by the player
8       * (the higher, the better). Ideally bounded between [-1, 1].
9       * @param gs - game state to evaluate and score.
10      * @param playerId  - id of the player we're evaluating the game for.
11      * @return - value of given state.
12      */
13     double evaluateState(AbstractGameState gs, int playerId);
14 }
```

In order to create your own heuristic for a game, you can write a class that implements the IStateHeuristic interface and implements the `evaluateState` method. In this method, you can analyze the contents of the game state (in the variable *gs*) and return a score for the game state that depends on the playerID receive: a game state may be better or worse depending on the perspective of different players (for example: who's winning vs. who's losing). The way in which this method is intended to be called is as follows:

```
1  double score = heuristic.evaluateState(gameState, gs.getCurrentPlayer());
```

where the state *gameState* is evaluated according to the player who makes the call ('id' retrieved by *gs.getCurrentPlayer()*).

An example of a state heuristic for the game Tic Tac Toe can be found in *games.tictactoe.TicTacToeHeuristic*. The snippet below shows a portion of this class (see the actual file for the full version):

```
public class TicTacToeHeuristic implements IStateHeuristic {
  double FACTOR_PLAYER = 0.8;
  double FACTOR_OPPONENT = 0.5;

  public double evaluateState(AbstractGameState gs, int playerId) {

    // Retrieve the current result
    TicTacToeGameState ttgs = (TicTacToeGameState) gs;
    Utils.GameResult playerResult = gs.getPlayerResults()[playerId];

    // and simply return 1 or -1 if it is over.
    if(playerResult == Utils.GameResult.LOSE)  return -1;
    if(playerResult == Utils.GameResult.WIN)   return 1;

    // Count how many lines of player characters + rest empty,
    // the more player characters the better
    int[] nPlayer = new int[ttgs.gridBoard.getWidth()];
    int[] nOpponent = new int[ttgs.gridBoard.getWidth()];

    // N rows + N columns + 2 diagonals
    double nTotalCount = nPlayer.length * 2 + 2;

    Token playerChar = TicTacToeConstants.playerMapping.get(playerId);

    // Check columns
    for (int x = 0; x < ttgs.gridBoard.getWidth(); x++)
        addCounts(countColumns(ttgs, x, playerChar), nPlayer, nOpponent);

    // Check rows
    for (int y = 0; y < ttgs.gridBoard.getHeight(); y++)
        addCounts(countRows(ttgs, y, playerChar), nPlayer, nOpponent);

    // Check diagonals
    // Primary
    addCounts(countPrimaryDiagonal(ttgs, playerChar),nPlayer,nOpponent);
    // Secondary
    addCounts(countSecondaryDiagonal(ttgs, playerChar),nPlayer,nOpponent);

    // Calculate scores, the more chars. for player, the higher the weight
    double pScore = 0, oppScore = 0;
    for (int i = 0; i < nPlayer.length; i++) {
        pScore += nPlayer[i]/nTotalCount * Math.pow(0.5, nPlayer.length-i);
        oppScore += nOpponent[i]/nTotalCount * Math.pow(0.3, nOpponent.length-i);
    }

    // The evaluation of this state factors the positions of
    // the player and the opponent.
    return pScore * FACTOR_PLAYER + oppScore * FACTOR_OPPONENT;
  }
}
```

Most games in TAG come with a heuristic implemented (like the one above). Given a state *gameState*, a call to gameState.getHeuristicScore(playerID) will **automatically** call the heuristic evaluation function of the game that is being played. The One Step Look Ahead agent above calls a heuristic function directly (*heuristic.evaluateState(gsCopy, playerID)*). In the following lab we will see how new heuristics can be defined and how to set them via agent parameters.

Exercise 8 Take a look at the TicTacToe heuristic (in the java file, *games.tictactoe.TicTacToeHeuristic*) and make sure you understand how the state evaluation is calculated. Remember that you can use the

debugger tool to inspect the values of the different values and to execute the code step by step. Ask in class if anything is not clear!

Dive Deeper 9 Repeat the last exercise of looking at the code of different heuristics classes to see how states can be evaluated in different games. Look at the following ones:

- Connect 4 (`games.connect4.Connect4Heuristic`)[a].
- Love Letter (`games.loveletter.LoveLetterHeuristic`).[b]
- Pandemic (`games.pandemic.PandemicHeuristic`).[c]

[a] This one is pretty simple, isn't it?
[b] More information about this game here: `https://en.wikipedia.org/wiki/Love_Letter_(card_game)`
[c] More information about this game here: `https://en.wikipedia.org/wiki/Pandemic_(board_game)`